Two sum

#include <vector>

#include <unordered\_map>

using namespace std;

class Solution {

public:

    vector<int> two Sum(vector<int>& nums, int target) {

        unordered\_map<int, int> seen; // value -> index

        for (int i = 0; i < nums.size(); ++i) {

            int complement = target - nums[i];

            if (seen.count(complement)) {

                return {seen[complement], i};

            }

            seen[nums[i]] = i;

        }

        return {}; // no solution found (should not happen if guaranteed one solution)

    }

};

Add Two numbers

class Solution {

public:

    ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

        ListNode dummy;          // dummy head for result list

        ListNode\* current = &dummy;

        int carry = 0;

        while (l1 != nullptr || l2 != nullptr || carry != 0) {

int sum = carry;

            if (l1 != nullptr) {

                sum += l1->val;

                l1 = l1->next;

            }

            if (l2 != nullptr) {

                sum += l2->val;

                l2 = l2->next;

            }

            carry = sum / 10;

            current->next = new ListNode(sum % 10);

            current = current->next;

        }

        return dummy.next;

    }

};

Longest Substring Without Repeating Characters

#include <unordered\_set>

#include <string>

using namespace std;

class Solution {

public:

    int lengthOfLongestSubstring(string s) {

        unordered\_set<char> seen;

        int left = 0, right = 0, maxLen = 0;

        int n = s.size();

        while (right < n) {

            if (seen.find(s[right]) == seen.end()) {

                // Character not seen → expand window

                seen.insert(s[right]);

                maxLen = max(maxLen, right - left + 1);

                right++;

            } else {

                // Character already in set → shrink window

                seen.erase(s[left]);

                left++;

            }

        }

        return maxLen;

    }

};

Median of Two Sorted Arrays

#include <vector>

#include <algorithm>

using namespace std;

class Solution {

public:

    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

        if (nums1.size() > nums2.size())

            return findMedianSortedArrays(nums2, nums1); // ensure nums1 is smaller

        int m = nums1.size();

        int n = nums2.size();

        int low = 0, high = m;

        while (low <= high) {

            int partitionX = (low + high) / 2;

            int partitionY = (m + n + 1) / 2 - partitionX;

            int maxLeftX = (partitionX == 0) ? INT\_MIN : nums1[partitionX - 1];

            int minRightX = (partitionX == m) ? INT\_MAX : nums1[partitionX];

            int maxLeftY = (partitionY == 0) ? INT\_MIN : nums2[partitionY - 1];

            int minRightY = (partitionY == n) ? INT\_MAX : nums2[partitionY];

            if (maxLeftX <= minRightY && maxLeftY <= minRightX) {

                if ((m + n) % 2 == 0)

                    return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2.0;

                else

                    return max(maxLeftX, maxLeftY);

            } else if (maxLeftX > minRightY) {

                high = partitionX - 1;

            } else {

                low = partitionX + 1;

            }

        }

        throw invalid\_argument("Input arrays are not sorted");

    }

};

Reverse Integer

#include <climits>  // for INT\_MAX and INT\_MIN

class Solution {

public:

    int reverse(int x) {

        long long rev = 0;  // use long long to detect overflow

        while (x != 0) {

            int digit = x % 10;

            x /= 10;

            rev = rev \* 10 + digit;

            if (rev > INT\_MAX || rev < INT\_MIN)

                return 0;  // overflow detected

        }

        return static\_cast<int>(rev);

    }

};

Merge Two Sorted Lists

class Solution {

public:

    ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) {

        ListNode dummy;             // dummy node to simplify list building

        ListNode\* tail = &dummy;    // pointer to the last node of merged list

        while (list1 != nullptr && list2 != nullptr) {

            if (list1->val < list2->val) {

                tail->next = list1;

                list1 = list1->next;

            } else {

                tail->next = list2;

                list2 = list2->next;

            }

            tail = tail->next;

        }

        // Attach the remaining part (only one list will be non-empty)

        tail->next = (list1 != nullptr) ? list1 : list2;

        return dummy.next;

    }

};